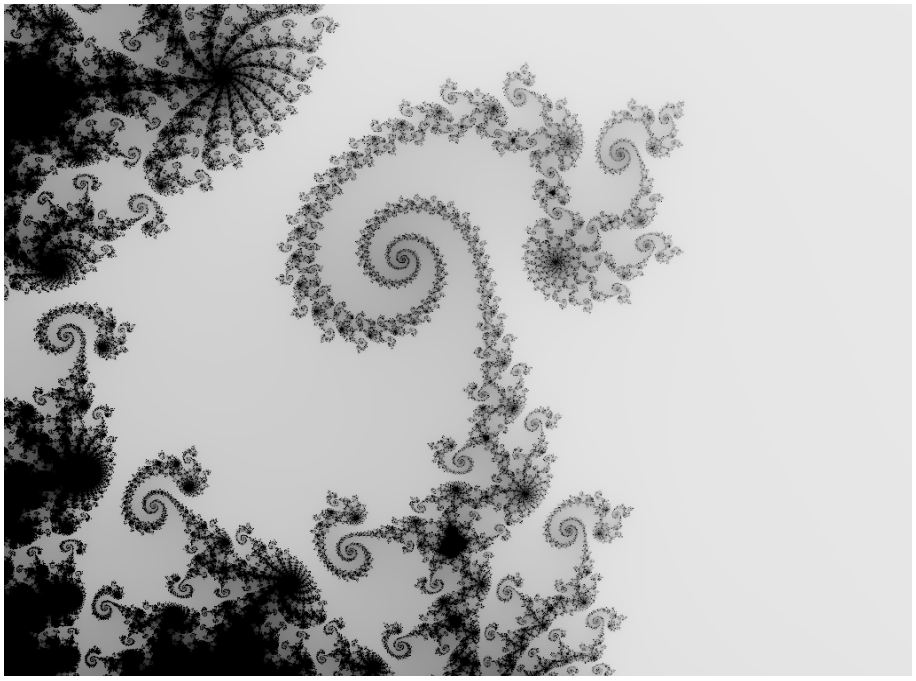
CS 789
Multiprocessor Programming

Parallelizing the Mandelbrot Computation.

School of Computer Science
Howard Hughes College of Engineering
University of Nevada, Las Vegas



1 Parallel Mandelbrot

Recall that in a Mandelbrot fractal, each pixel in the final image can be computed independently of all the other pixels. If we had a large shared memory multi processor with $M * N$ processors, we could compute a picture of size $M * N$ in constant time assigning one pixel in the final image to each of the $M * N$ processors.

This sort of fine grained parallelism does not lend itself to message passing. Mainly because of the large start up cost of actual processes, but also because of the overhead of passing messages and also because we do not have that many processors available. (For a 2000 by 1500 pixel picture we would need 30,0000 processors.)

So we need some other way of parallelizing this problem. Luckily, since every pixel can be computed independently, it is an easy task to do. Problems that parallelize in such an easy way, i.e., without dependences on previous or neighbour values, are referred to as **embarrassingly parallel**.

This also means that if we distribute the problem in a good/smart fashion, we can achieve nearly perfect load balancing. This means that all the processes working on the problem will all take the same time to finish.

However, it is possible to distribute the work in an 'evenly' manner that does not give a good load balancing. To achieve 100% perfect load, we would actually need the picture that we are computing, so that defeats the purpose of trying to compute it.

2 What to Do

- Implement a parallel version of the Mandelbrot program using MPI. There should be one master process and a number of slave processes. It is up to you to decide if you want to distribute the problem by using broadcast or multiple unicast. Also, you decide if you want to use `MPI_IRecv` or just the regular `MPI_Recv`.

- Try with at least 2 different data distribution schemes, that is, provide 2 versions of your program, with different ways of partitioning the problem. Explain your choices and try to predict how well it will load balance.

- If you didn't implement it, try to describe the most efficient way to distribute this problem in order to achieve good load balance. Can a general optimal solution for distributing the load evenly be achieved?

- Run your 2 versions on 2, 3, 4, ... nodes a number of times, and record their time.Provide plots that show the runtime of your parallel versions of the program (one plot for each program, time on the Y-axis and the number of processors on the X-axis). In order to time your program see the following section.

- Compute speedups and provide a table or a graph that shows your speedup. You can use my sequential version for measuring T_{seq}.

- Using timers (see the next section), for your best implementation, measure T_{comm}, T_{comp} and T_{IO} (the file IO time for the master process), and comment on them.

If you are running on a large number of processors, you might want to make the picture you produce larger.

3 Timing

To time certain parts of your code you can do the following:

```
#include <sys/time.h>
.
.

void printTime(int rank, struct timeval t1, struct timeval t2) {
  long long l;
  long secs, usecs;

  l = t2.tv_sec*1000000+t2.tv_usec-(t1.tv_sec*1000000+t1.tv_usec);

  secs = l/1000000;
  usecs = l%1000000;

  printf("%d: (%d:%d -> %d:%d (%d:%d)\n",rank,t1.tv_sec,t1.tv_usec,
         t2.tv_sec,t2.tv_usec,secs,usecs);
}

int main(...) {
  struct timeval t1, t2;

  gettimeofday(&t1,NULL);

  /* Code that you want to measure */

  gettimeofday(&t2,NULL);
  printTime(rank, t1, t2)
}
```

Grab the time in `t1` when the code that you want to measure starts, and grab it again in `t2` and use `printTime` to compute the actual time.

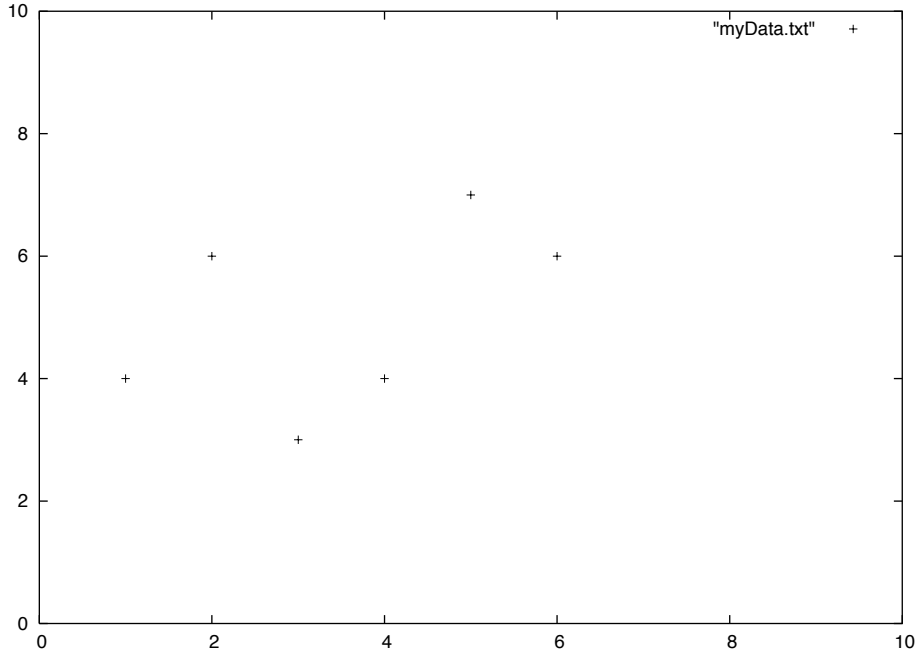
4 Using Gnuplot to Create Graphs

The tool 'gnuplot' can be used to create nice graphs of runtime and speedup. You need an input file containing the points you want to plot, pairs of numbers representing the X, Y (and Z if needed) coordinates. The following file

```
1 4
```

```
2 6
3 3
4 4
5 7
6 6
```

will produce this graph



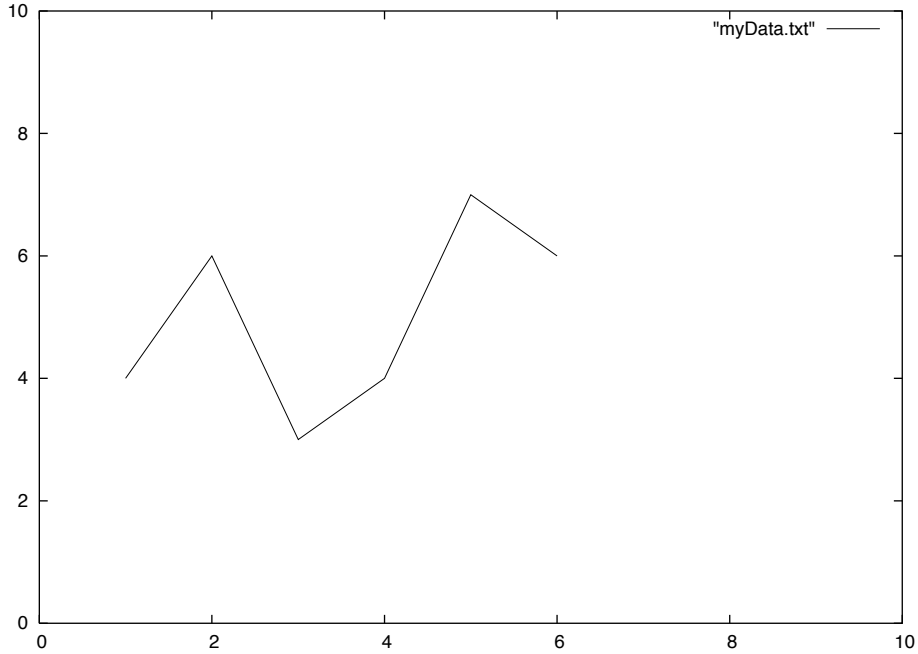
if printed in the following way (the data is in a file called myData.txt):

```
plot [0:10] [0:10]  ‘‘myData.txt’’
```

and

```
plot [0:10] [0:10]  ‘‘myData.txt’’ with lines
```

will produce



If you want to produce a picture that you can print out or include as a picture in a latex file here is the list of commands:

```
set term postscript eps
set output ‘‘outfile.eps’’
plot [0:10] [0:10]  ‘‘myData.txt’’
```

You can now include this postscript figure in your latex code using the `psfig` package and the command \`psfig`{`figure=outfile.eps`} command.